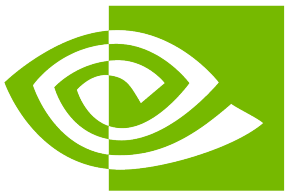

NVIDIA Triton Inference Server User Guide for AR and VFX SDKs

Release 1.2.2

NVIDIA Corporation

Sep 10, 2026

Contents

Download this guide as a PDF

The NVIDIA Triton inference server (https://developer.nvidia.com/nvidia-triton-inference-server) is software that helps to deploy AI features in the cloud at scale. The Triton-enabled NVIDIA Augmented Reality (AR) and Video Effects (VFX) SDKs are well suited for servers and microservices because the SDKs use several Triton features to deliver

- High throughput and resource utilization through dynamic batching, batched inference, and concurrent model execution.
- Multistream support to process multiple video streams concurrently.
- MultiGPU and MIG support.

With the Triton-enabled SDKs, the AR and VFX SDK features run fully on a Triton server and the SDK library provides C APIs for the user's application to communicate with the server via gRPC. Because the server supports multi-stream processing, several applications can send workload to a single server at the same time, each in turn requesting processing on more than one input video stream.

The following features are included in the Triton-enabled AR SDK:

- Face Detection
- Facial Landmark Detection
- Eye Contact
- LipSync
- Active Speaker Detection
- 3D Body Pose Estimation

The following features are included in the Triton-enabled VFX SDK:

- AI Green Screen
- Background Blur
- Video Relighting

The image data is transferred between the application and the server as raw frames using gRPC, or using shared memory in cases when the server and the application run on the same machine. This limits the usage of the SDK for real-time processing when the server and the application are on separate machines connected by a network because sending raw frames over gRPC has high latency.

Chapter 1. Performance Reference

The following table lists multistream performance of Triton-enabled SDKs in comparison with the performance of the non-Triton organic SDKs.

AR SDK - Face Detection

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	75	136	81%
A40	114	180	58%
L40	138	237	72%
B40	138	299	116%

AR SDK - Facial Landmark Detection

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	36	98	172%
A40	54	150	177%
L40	75	156	108%
B40	60	276	360%

AR SDK - Eye Contact

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	8	13	62%
A40	11	23	109%
L40	16	26	62%
B40	19	42	121%

AR SDK - LipSync

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	0	0	NA
A40	1	1	NA
L40	2	2	NA
B40	2	2	NA

AR SDK - Active Speaker Detection - Single Speaker Use Case

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	1	1	NA
A40	3	3	NA
L40	5	5	NA
B40	4	4	NA

AR SDK - 3D Body Pose Estimation - Single body

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	0	0	NA
A40	0	0	NA
L40	1	1	NA
B40	2	1	NA

VFX SDK - AI Green Screen

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	13	12	-7%
A40	27	32	18%
L40	44	54	23%
B40	37	57	54%

VFX SDK - AIGS Relighting

	Throughput		
GPU	Organic SDK	Triton-enabled SDK	Throughput gain
T4	0	0	NA
A40	1	2	100%
L40	4	9	125%
B40	5	12	140%

Note

The input video resolution is 720p. AI Green Screen is using Performance mode. Facial Landmark Detection is computed with 126 landmark points detected on face in Performance mode.

Face Detection, Facial Landmark Detection, and Eye Contact have temporal flags on. The server and the client are on the same machine, and the data transfer uses CUDA Shared Memory.

Throughput is defined as the maximum number of streams to achieve 30 FPS in real time.

1.1. Hardware and Software Requirements

The Triton-enabled VFX and AR SDKs are supported only on Linux. For the supported hardware and software, refer to the following pages:

- VFX SDK: Get Started on Linux.
- AR SDK: Get Started on Linux.

To download or build the required software for Triton server and Triton client, refer to the *Installation* section of this guide.

1.2. Installation

The Triton-enabled VFX and AR SDKs consist of two components: the SDK package and the server. You must install the SDK package before installing the server.

1.2.1. SDK Package

The SDK is delivered as a `.tgz` package, which is a compressed tar format.

The package contains the AR or VFX SDK library and header files, which need to be extracted to the local system. In the SDK package, see the `README_quickstart.md` file for more detailed instructions about setting up the environment to use the SDK.

- Install the SDK:

AR SDK

```
$ sudo tar -xvf ARSDK_linux_<version>.tgz -C /usr/local
```

VFX SDK

```
$ sudo tar -xvf VFXSDK_linux_<version>.tgz -C /usr/local
```

- Build and install the Triton client library and dependencies:

AR SDK

```
$ sudo bash /usr/local/ARSDK/share/build_triton_client_lib.sh -i /usr/
↪local/ARSDK/lib
```

VFX SDK

```
$ sudo bash /usr/local/VideoFX/share/build_triton_client_lib.sh -i /usr/
↪local/VideoFX/lib
```

1.2.2. Server

The server is delivered as a `.tgz` package and contains the SDK libraries. The server runs inside the Triton Docker container.

As prerequisites, you must install the appropriate NVIDIA Graphics Driver for Linux, Docker Engine, and the NVIDIA Container Toolkit.

- NVIDIA Graphics Driver for Linux
  - AR SDK: Software Requirements.
  - VFX SDK: Software Requirements.
- Docker
  - Ubuntu, CentOS, or Debian: Install Docker Engine.
  - Rocky Linux 8: Docker - Install Engine.
  - Rocky Linux 9: Docker - Install Engine.
- NVIDIA Container Toolkit: Follow the installation and Docker configuration instructions in Installing the NVIDIA Container Toolkit.

To install the server package, untar the package to a location of your choice. The following example shows how to install the package in your home directory and change the directory to the installation location:

AR SDK

```
$ tar -xvf ARSDK_triton_<version>.tgz -C ~
$ cd ~/ARSDK-triton-server
```

VFX SDK

```
$ tar -xvf VFXSDK_triton_<version>.tgz -C ~
$ cd ~/VideoFX-triton-server
```

You must initialize the model repository by adding the model files. The following example shows how to initialize the model repository by specifying your NGC account with organization and team, AI feature, and model repo from anywhere within the appropriate root folder (such as `ARSDK-triton-server` or `VideoFX-triton-server`):

AR SDK

```
$ export NGC_CLI_API_KEY=<your_API_key>
$ sudo --preserve-env=NGC_CLI_API_KEY ./init_model_repo.sh --ngc-org org --
↪ngc-team team -f all --modelrepo ~/ARSDK-triton-server/arsdk_model_repo
```

VFX SDK

```
$ export NGC_CLI_API_KEY=<your_API_key>
$ sudo --preserve-env=NGC_CLI_API_KEY ./init_model_repo.sh --ngc-org org --
↪ngc-team team -f all --modelrepo ~/VideoFX-triton-server/vfxsdk_model_repo
```

Run the following script to download the Docker image and run the Triton server:

```
$ sudo bash run_triton_server.sh
```

The script runs the Triton server and loads the SDK features. If successful, the server displays messages saying the gRPC and metric services started and blocks the terminal as shown:

```
I0411 22:05:33.459919 1 grpc_server.cc:4819] Started GRPCInferenceService at
↪0.0.0.0:8001
I0411 22:05:33.502086 1 http_server.cc:184] Started Metrics Service at 0.0.0.
↪0:8002
```

Check the displayed messages and verify that the following information appears, showing that all models are loaded.

AR SDK

```
+---------------------------------+---------+--------+
| Model                           | Version | Status |
+---------------------------------+---------+--------+
| 3DBodyPoseEstimation            | 1       | READY  |
| ActiveSpeakerDetection          | 1       | READY  |
| ActiveSpeakerDetectionMultiView | 1       | READY  |
| FaceBox                         | 1       | READY  |
| FaceKeypoints126Mode0           | 1       | READY  |
| FaceKeypoints126Mode1           | 1       | READY  |
| FaceKeypoints68Mode0            | 1       | READY  |
| FaceKeypoints68Mode1            | 1       | READY  |
| GazeRedirectionKey126           | 1       | READY  |
| GazeRedirectionKey68            | 1       | READY  |
| Lipsync                         | 1       | READY  |
```

(continues on next page)

(continued from previous page)

```
| LipsyncU16                      | 1       | READY  |
| LipsyncU16_DE                   | 1       | READY  |
| LipsyncU16_ES                   | 1       | READY  |
| LipsyncU16_FR                   | 1       | READY  |
| Lipsync_DE                      | 1       | READY  |
| Lipsync_ES                      | 1       | READY  |
| Lipsync_FR                      | 1       | READY  |
+---------------------------------+---------+--------+
```

VFX SDK

```
+----------------------------+---------+--------+
| Model                      | Version | Status |
+----------------------------+---------+--------+
| AigsMode0                  | 1       | READY  |
| AigsMode1                  | 1       | READY  |
| AigsMode2                  | 1       | READY  |
| AigsMode3                  | 1       | READY  |
| BackgroundBlur             | 1       | READY  |
| CompAigsRelightingBkgMode1 | 1       | READY  |
| CompAigsRelightingBkgMode3 | 1       | READY  |
| CompAigsRelightingPrjMode1 | 1       | READY  |
| CompAigsRelightingPrjMode3 | 1       | READY  |
| CompAigsRelightingSrcMode1 | 1       | READY  |
| CompAigsRelightingSrcMode3 | 1       | READY  |
| RelightingProj0Mode1       | 1       | READY  |
| RelightingProj1Mode1       | 1       | READY  |
+----------------------------+---------+--------+
```

1.3. Build the Sample Applications

Triton client applications are included in the sample apps repository available at https://github.com/NVIDIA-Maxine/AR-SDK-Samples for the AR SDK and https://github.com/NVIDIA-Maxine/VFX-SDK-Samples for the VFX SDK.

Follow the instructions in `README.md` in the “Triton - Linux Only” section.

1.4. Design

The Triton-enabled SDKs have two components: the server and the SDK library. The server contains the entire feature pipeline and is responsible for the end-to-end execution of the feature. The SDK library is responsible for sending the input data from the user’s application to the server, running the feature on the server, and providing output back to the user’s application.

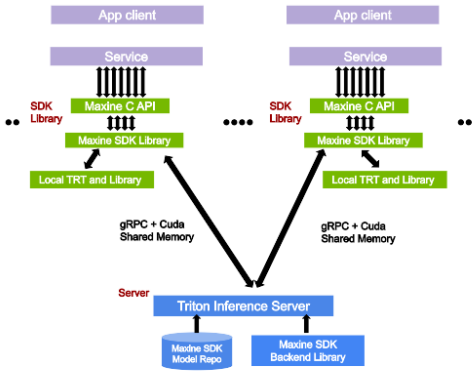

Fig. 1: Figure 1. Triton-enabled SDK design.

1.5. Server

The server consists of a Triton Inference Server (https://developer.nvidia.com/nvidia-triton-inference-server), the Maxine SDK model repository, and the Maxine SDK backend library.

The Triton-enabled AR and VFX SDKs use the following features in the Triton Inference Server to deliver higher throughput and multistream support.

- **Dynamic batching** combines separate requests into batches for concurrent execution during runtime to provide higher throughput. More details about dynamic batching can be found here:
  - https://github.com/triton-inference-server/tutorials/tree/main/Conceptual_Guide/Part_2-improving_resource_utilization#what-is-dynamic-batching
  - https://github.com/triton-inference-server/server/blob/main/docs/user_guide/batcher.md#dynamic-batcher
- **Sequence batching** provides support for processing multiple input video streams concurrently. More details about sequence batching can be found here:
  - https://github.com/triton-inference-server/server/blob/main/docs/user_guide/batcher.md#sequence-batcher
- **Concurrent execution** on a single GPU or multi-GPU systems. Triton server can create multiple instances of the feature on a single GPU or on multiple GPUs, which can aid in parallel processing of the requests. More details about concurrent execution can be found here:
  - https://github.com/triton-inference-server/tutorials/tree/main/Conceptual_Guide/Part_2-improving_resource_utilization#concurrent-model-execution
  - https://github.com/triton-inference-server/server/blob/main/docs/user_guide/model_execution.md#concurrent-model-execution

The Maxine SDK backend library and the Maxine SDK model repository implement the Maxine feature on the Triton server. Maxine SDK model repository contains models and configuration files.

The server is supplied with default configuration files, which can be used as it is. We recommended that users do not edit the tensor names, type, shape, or the sequence batching and ensemble architecture in the configuration files. However, some of the parameters in the configuration files can be modified to optimize performance and to enable or disable certain features, as discussed later. For more details about the Triton model configuration files, refer to Model Configuration.

1.5.1. Configuring AR SDK Server

Each feature is implemented as a folder in the model repository. The following table lists the location for the configuration file for each feature.

Feature	Configuration File Location
Face Detection	FaceBox
Face Keypoints, 68 Keypoints, mode 0	FaceKeypoints68Mode0
Face Keypoints, 68 Keypoints, mode 1	FaceKeypoints68Mode1
Face Keypoints, 68 Keypoints, mode 0	FaceKeypoints126Mode0
Face Keypoints, 68 Keypoints, mode 1	FaceKeypoints126Mode1
Gaze Redirection, 68 Keypoints	GazeRedirectionKey68
Gaze Redirection, 126 Keypoints	GazeRedirectionKey126
LipSync, U8 mode	Lipsync
LipSync, U16 mode	LipsyncU16
LipSync, U8 mode, German	Lipsync_DE
LipSync, U16 mode, German	LipsyncU16_DE
LipSync, U8 mode, French	Lipsync_FR
LipSync, U16 mode, French	LipsyncU16_FR
LipSync, U8 mode, Spanish	Lipsync_ES
LipSync, U16 mode, Spanish	LipsyncU16_ES
Active Speaker Detection	ActiveSpeakerDetection
Active Speaker Detection, multi-view mode	ActiveSpeakerDetectionMultiView
3D Body Pose Estimation	3DBodyPoseEstimation

The following parameters can be modified in the configuration file:

- Maximum batch size

  The property **max_batch_size** in the configuration file sets the maximum size of the batch Triton uses with the dynamic batcher. We recommend that this parameter be set to a value equal to the expected number of active video streams.

- Dynamic batching parameters

  The dynamic batching can be optimized by setting the following properties.

  - `max_candidate_sequences`: The maximum number of possible concurrent video streams.
  - `max_queue_delay_microseconds`: The amount of time, in microseconds, that the dynamic batcher will wait to complete the batch.
  - `max_sequence_idle_microseconds`: The amount of time, in microseconds, an idle input video stream is kept active.

- Instance group

  The instance group property can be used to create multiple instances of the feature on the Triton, either on the same GPU or on multiple GPUs. For more details, refer to Instance Groups. Note that the `kind` field should always be set to GPU.

1.5.2. Configuring VFX SDK Server

The AI Green Screen is implemented as a Triton ensemble (https://github.com/triton-inference-server/server/blob/main/docs/user_guide/ensemble_models.md#ensemble-models). In the model repository, the folders **AigsEnsembleMode0** and **AigsEnsembleMode1** have the ensemble for AI Green Screen mode 0 and mode 1. The corresponding models and configuration files for mode 0 and mode 1 are in **AigsStatefulModelMode0** and **AigsStatefulModelMode1**.

The following parameters may be modified in the configuration file in *AigsStatefulModelMode0* and *AigsStatefulModelMode1 folders*:

- Dynamic batching parameters

  The dynamic batching can be optimized by using the following properties.

  - max_candidate_sequences: it sets the maximum number of possible concurrent video streams
  - max_queue_delay_microseconds: the amount of time the dynamic batcher will wait to complete the batch in microseconds
  - max_sequence_idle_microseconds: it sets the time in microseconds an idle input video stream is kept active

- Instance group

  The instance group property can be used to create multiple instances of the feature on the Triton, either on the same GPU or on multiple GPUs. For more details, refer to Instance Groups. Note that the `kind` field should always be set to GPU.

1.6. SDK Library

The SDK library consists of the Maxine SDK C API and Library, which is used by the application to communicate with the server. Several applications can connect to the same Triton server where the individual requests are dynamically batched for concurrent processing. Each application can send one request or a batch of requests simultaneously.

The SDK library communicates with the server using the gRPC. The video data between the server and the SDK library is transferred as raw frames. We recommende that the server and the application be run on the same machine, not separate machines connected over a network, because it is not possible to achieve real-time video processing by transferring raw video frames using gRPC. This is not a problem when the server and the application are on the same machine. Additionally, on the same machine, applications can choose the option to share the image buffers between server and SDK library using CUDA shared memory, which can achieve significantly faster data transfer. For more information, refer to Shared-Memory Extension.

Depending on what APIs are called, the SDK library can also run the features locally, rather than on a server. In this case, the applications will not be able to utilize the benefits of the Triton server. The details about the API usage is explained in the programming guide section.

1.7. Recommendations

We recommend that the Triton-enabled SDK is integrated with the service as shown in Figure 1. The user's service application, the SDK library, and the server reside on the same machine. The video data transfer between the service application and the server is done using CUDA shared memory via the

SDK library. Multiple service applications can send requests to the same Triton server, or a single service application can create several threads that send requests to the same Triton server. Even within the same thread, the service application can send a batch of requests to the same Triton server using the Maxine SDK batching APIs.

1.8. Programming Guides

Several new APIs and data types are added to the Maxine SDK C API to enable running features on Triton. It includes APIs related to communication with the server and batching. The following sections explain the usage of these APIs. For detailed information regarding the usage in an application, refer to `TritonClientApp` in the SDKs.

1.9. AR SDK

1.9.1. Creating a Triton Object and Connecting to an ARSDK Enabled Triton Server

`NvAR_TritonServer` is a predefined type in the SDK, which represents a connection to the server. The first step is to create an instance of this type and call the `NvAR_ConnectTritonServer` to connect to a server. The `NvAR_ConnectTritonServer` function accepts a string indicating the URL (IP and the port) of the gRPC service on the server. In the same application, it is possible to connect to multiple servers by creating multiple instances.

```
NvCV_Status nvcv_err = NVCV_SUCCESS;

NvAR_TritonServer server_handle;

char server_url[] = "127.0.0.1:8001";

nvcv_err = NvAR_ConnectTritonServer(server_url, &server_handle);
```

1.9.2. Creating a Feature Instance

`NvAR_FeatureHandle` is a predefined type in the SDK, which represents a feature. To instantiate a handle for a feature to run on a server, `NvAR_CreateTriton` API is called. This is different from `NvAR_Create` API which creates a feature to be run locally. Only feature instances created using `NvAR_CreateTriton` can run features on the server and use the batching APIs. If a feature handle is instantiated using `NvAR_Create`, its usage is limited to exactly the same as the non-Triton enabled SDKs.

The `NvAR_CreateTriton` API takes feature ID as input which indicates what feature to instantiate. It could be NvAR_Feature_FaceDetection, NvAR_Feature_LandmarkDetection, or NvAR_Feature_GazeRedirection. The next step is to assign the feature instance to a server instance using `NvAR_SetTritonServer`. Several feature instances can be assigned to a single server instance, but a single feature instance can be associated with only one server instance.

```
NvAR_FeatureHandle feature_handle;

nvcv_err = NvAR_CreateTriton(NvAR_Feature_FaceDetection, &feature_handle);

nvcv_err = NvAR_SetTritonServer(feature_handle, server_handle);
```

1.9.3. Getting and Setting Feature Properties

Once the feature instance is instantiated and assigned to a server, feature configurations may be set using the setters and verified using the getters. For example, the following code sets the temporal flag. The features support the same configurations as the non-Triton SDKs.

```
nvcv_err = NvAR_SetU32(feature_handle, NvAR_Parameter_Config(Temporal),
↪0xFFFFFFFF);
```

1.9.4. Loading the Feature

`NvAR_Load` is called to initialize the feature after the configurations have been set.

```
nvcv_err = NvAR_Load(feature_handle);
```

1.9.5. Creating State Objects

The SDK uses state objects to identify and track the input video stream in a batched input. One state object is associated with one input video stream. Therefore, if the application intends to process N video streams, N state objects must be created. The SDK specifies `NvAR_StateHandle` type for the state objects which are instantiated using `NvAR_AllocateState` as shown below. Note that even if the application is processing one video stream, it is necessary to create one state object to represent that video stream.

```
int num_input_video_streams = 10;

std::vector<NvAR_StateHandle> state_handles(num_input_video_streams);

for (int i=0; i<num_input_video_streams; i++) {

   nvcv_err = NvAR_AllocateState(feature_handle, &state_handles[i]);

}
```

It is not necessary to create all the state objects at once, new state objects may be created during runtime as needed.

1.9.6. Setting Input and Output

The process of setting input and output buffers for the features is the same as the non-Triton enabled SDKs. The only difference is that when the application is trying to send a batch of input (more than one video stream concurrently) to the server, it is necessary to provide batched buffers.

1.9.6.1 Batched NvCVImage Object Buffers

A batched NvCVImage points to a contiguous buffer that contains multiple images with the same structure. The Triton enabled SDK library will transfer NvCVImage to and back from the server using gRPC if the NvCVImage buffer is allocated on the CPU and using CUDA shared memory if the NvCVImage buffer is allocated on the GPU.

There are tighter restrictions on the type of images supplied to the Triton Client, relative to the native SDK:

- The pixel format is restricted to BGR, BGRA, and A; RGB and RGBA are not accommodated.
- The images must have a zero-gap pitch. This can be achieved by specifying 1 as the alignment, when calling the NvCVImage constructor or NvCVImage_Alloc, NvCVImage_Realloc.

1.9.6.2 Batch Utilities

The following utility functions in BatchUtilities.cpp help you to work with image batches:

- AllocateBatchBuffer() can be used to allocate a buffer for a batch of images.
- NthImage() can be used to set a view into the nth image in a batched buffer.
- ComputeImageBytes()can be used to determine the number of bytes for each image to advance the pixel pointer from one image to the next.
- TransferToNthImage()makes it easy to call NvCVImage_Transfer() to set one of the images in a batch.
- TransferFromNthImage()makes it easy to call NvCVImage_Transfer() to copy one of the images in a batch to a regular image.
- TransferToBatchImage()transfers multiple images from different locations to a batched image.
- TransferFromBatchImage()can be used to retrieve the images in a batch to different images in different locations.
- TransferBatchImage() transfers all images in a batch to another compatible batch of images.

The last three functions can also be accomplished by repeatedly calling the Nth image APIs, but the source code illustrates an alternative method of accessing images in a batch.

1.9.6.3 Allocation of Batched Buffers

To allocate batched buffers, call the AllocateBatchBuffer() function, which will allocate an image that is N times taller than the prototypical image.

> **Note**
>
> The allocation cannot always be interpreted this way, especially if the pixels are planar.

The purpose of this function is mainly to provide storage and then dispose of this storage when the NvCVImage goes out of scope or its destructor is called.

You can use your own method to allocate the storage for the batched images. The image that the AllocateBatchBuffer() yields is only used for bookkeeping and is never used in any of the SDK APIs.

Note

The SDK APIs only require an NvCVImage descriptor for the first image.

1.9.6.4 Setting the Batched Images

The API only takes the image descriptors for the first image in a batch. The following sample allocates the src and dst batch buffers and sets the input and outputs via the views of the first image in each batch buffer.

```
NvCVImage srcBatch, dstBatch, nthSrc, nthDst;
AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                    ...);
AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                    ...);
NthImage(0, srcHeight, &srcBatch, &nthSrc);
NthImage(0, dstHeight, &dstBatch, &nthDst);
NvAR_SetObject(effect, NvAR_Parameter_Output(Image), &nthSrc,
↪sizeof(NvCVImage));
NvAR_SetObject(effect, NvAR_Parameter_Output(Image), &nthDst,
↪sizeof(NvCVImage));
```

Because the image descriptors are copied into the SDK, this can be simplified to the following:

```
NvCVImage srcBatch, dstBatch, nth;
AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                    ...);
AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                    ...);
NvAR_SetObject(effect, NvAR_Parameter_Input(Image),
               NthImage(0, srcHeight, &srcBatch, &nth), sizeof(NvCVImage));
NvAR_SetObject(effect, NvAR_Parameter_Output(Image),
               NthImage(0, dstHeight, &dstBatch, &nth), sizeof(NvCVImage));
```

The other images in the batch are computed by advancing the pixel pointer by the size of each image.

The other aspect of setting the batched images is determining how to set the pixel values. Each image in the batch is accessible by calling the NthImage() function:

```
NthImage(n, imageHeight, &batchImage, &nthImage);
```

You can then use the same techniques that were used for other NvCVImages on the recently initialized nthImage view. As previously suggested, NthImage() is just a thin wrapper around NvCVImage_InitView() and can be used instead. The NvCVImage_Transfer() functions can be used to copy pixels to and from the batch.

1.9.6.5 Batched bounding boxes

Batched bounding boxes is an array of `NvAR_BBoxes` structure, where each item's boxes field points to an array of `NvAR_Rect`. It can be set as follows:

```
unsigned batch_size = 3;
```

(continues on next page)

(continued from previous page)

```
std::vector<NvAR_BBoxes> output_bboxes(batch_size);

std::vector<std::vector<NvAR_Rect>> output_bboxes_data(batch_size);

for (unsigned i = 0; i < batch_size; i++) {

   output_bboxes_data[i].resize(25);

   output_bboxes_data[i].assign(25, { 0.f, 0.f, 0.f, 0.f });

   output_bboxes[i].boxes = output_bboxes_data[i].data();

   output_bboxes[i].max_boxes = kMaxBoxes;

   output_bboxes[i].num_boxes = 1;

}
```

Batched bounding boxes type is set in the SDK by providing the starting address of the buffer:

```
NvAR_SetObject(feature_handle, NvAR_Parameter_Output(BoundingBoxes),
output_bboxes.data(), sizeof(NvAR_BBoxes)))
```

1.9.6.6 Batched float, NvAR_Point2f, NvAR_Point3f, and NvAR_Quaternion

Batched `float`, `NvAR_Point2f`, `NvAR_Point2f`, and `NvAR_Quaternion` type buffers are contiguous arrays big enough to hold all of the input or the output in the batch.

They can be allocated as follows:

```
unsigned batch_size = 3;

std::vector<float> confidence(CONFIDENCE_SIZE*batch_size);
//one item inthe batch is CONFIDENCE_SIZE big

std::vector<NvAR_Point2f> keypoints_2d(KEYPOINTS2D_SIZE*batch_size);
//one item in the batch is KEYPOINTS2D_SIZE big

std::vector<NvAR_Point3f> keypoints_3d(KEYPOINTS3D_SIZE*batch_size);
//one item in the batch is KEYPOINTS3D_SIZE big

std::vector<NvAR_Quaternion> pose(batch_size);
// One item in the batch is of size 1
```

They are set in the SDK by providing the starting address of the buffer.

Example:

```
nvcv_err = NvAR_SetObject(feature_handle, NvAR_Parameter_Output(Pose),
pose.data(), sizeof(NvAR_Quaternion));
```

1.9.7. Setting a Batch of State Objects

The SDK uses batch size parameter, `NvAR_Parameter_Config(BatchSize)`, to determine the number of items in the input/output buffers and the batch of states parameter, `NvAR_Parameter_InOut(State)`, to determine which item in the batched input and output buffer belongs to which input video stream. It is necessary to set these parameters before the `NvAR_Run` call.

A batch of states array is an array of `NvAR_StateHandle` pointers. In the following example, a batch size of 3 is used and a batch of states is created with pointers to the p-th, the q-th and the r-th items in the `state_handles` array. This signifies that all input/output buffers contain data in the same order. Since each state corresponds to an input video stream, it signifies that the 0-th, the 1-st, and the 2-nd items in the buffers belong to the p-th, the q-th and the r-th input video streams.

A batch cannot contain more than one item from a single input video stream, so a batch of states should not have duplicate state objects.

Even when processing only one video, it is necessary to create a batch of states array with one item and set the batch size to 1.

```
unsigned batch_size = 3;

NvAR_StateHandle\* batch_of_states[] = {&state_handles[p],
&state_handles[q], &state_handles[r]};

nvcv_err = NvAR_SetU32(feature_handle, NvAR_Parameter_Config(BatchSize),
batch_size);

nvcv_err = NvAR_SetObject(feature_handle, NvAR_Parameter_InOut(State),
batch_of_states, batch_size);
```

The `NvAR_Parameter_Config(BatchSize)` and the `NvAR_Parameter_InOut(State)` can be changed before every `NvAR_Run` call.

```
The size of the batch need not be equal to the total number of input video
↪streams.
```

1.9.8. Running the Feature

`NvAR_Run` runs the feature on the server. The call is asynchronous and non-blocking.

```
nvcv_err = NvAR_Run(feature_handle);
```

1.9.9. Waiting for the Result

`NvAR_SynchronizeTriton` will block the execution and wait until the results are available after the `NvAR_Run` call. When this function exits, the results will be populated in the output buffers.

```
nvcv_err = NvAR_SynchronizeTriton(feature_handle);
```

1.9.10. Destroying the State Objects

`NvAR_DeallocateState` destroys the state objects.

```
for (int i=0; i<num_input_video_streams; i++) {

   nvcv_err = NvAR_DeallocateState(feature_handle, state_handles[i]);

}
```

1.9.11. Destroying the Feature Instance

`NvAR_Destroy` destroys the feature instance.

```
nvcv_err = NvAR_Destroy(feature_handle);
```

1.9.12. Destroying the Triton Object

`NvAR_DisconnectTritonServer` disconnects from the server and destroys the server object.

```
nvcv_err = NvAR_DisconnectTritonServer(server_handle);
```

1.10. VFX SDK

1.10.1. Creating a Triton Object and Connecting to an ARSDK Enabled Triton Server

`NvVFX_TritonServer` is a predefined type in the SDK, which represents a connection to the server. The first step is to create an instance of this type and call the `NvVFX_ConnectTritonServer` to connect to a server. The `NvVFX_ConnectTritonServer` function accepts a string indicating the URL (IP and the port) of the gRPC service on the server. In the same application, it is possible to connect to multiple servers by creating multiple instances.

```
NvCV_Status nvcv_err = NVCV_SUCCESS;

NvVFX_TritonServer server_handle;

char server_url[] = "127.0.0.1:8001";

nvcv_err = NvVFX_ConnectTritonServer(server_url, &server_handle);
```

1.10.2. Creating a Feature Instance

`NvVFX_Handle` is a predefined type in the SDK, which represents a feature. To instantiate a handle for a feature to run on a server, `NvVFX_CreateEffectTriton` API is called. This is different from the `NvVFX_CreateEffect` API which creates a feature to be run locally. Only feature instances created

using NvVFX_CreateEffectTriton can run features on the server and use the batching APIs. If a feature handle is instantiated using NvVFX_CreateEffect, its usage is limited to exactly the same as the non-Triton enabled SDKs.

The NvVFX_CreateEffectTriton API takes feature ID as input which indicates what feature to instantiate. For AI Green Screen, the feature ID is NVVFX_FX_GREEN_SCREEN. The next step is to assign the feature instance to a server instance using NvVFX_SetTritonServer. Several feature instances may be assigned to a single server instance, however one feature instance may be associated with only one server instance.

```
NvVFX_Handle feature_handle;

nvcv_err = NvVFX_CreateEffectTriton(NVVFX_FX_GREEN_SCREEN, &feature_handle);

nvcv_err = NvVFX_SetTritonServer(feature_handle, server_handle);
```

1.10.3. Getting and Setting Feature Properties

After the feature instance is instantiated and assigned to a server, feature configurations can be set using the setters and verified using the getters. For example, the following code sets the mode. The features support the same configurations as the non-Triton SDKs.

```
nvcv_err = NvVFX_SetU32(feature_handle, NVVFX_MODE, 0);
```

1.10.4. Loading the Feature

NvVFX_Load is called to initialize the feature after the configurations have been set.

```
nvcv_err = NvVFX_Load(feature_handle);
```

1.10.5. Creating State Objects

The SDK uses state objects to identify and track the input video stream in a batched input. One state object is associated with one input video stream. Therefore, if the application intends to process N video streams, N state objects must be created.

The SDK specifies NvVFX_StateObjectHandle type for the state objects which are instantiated using NvVFX_AllocateState as shown below. Note that even if the application is processing one video stream, it is necessary to create one state object to represent that video stream.

```
int num_input_video_streams = 10;

std::vector<NvVFX_StateObjectHandle>
state_handles(num_input_video_streams);

for (int i=0; i<num_input_video_streams; i++) {

   nvcv_err = NvVFX_AllocateState(feature_handle, &state_handles[i]);

}
```

It is not necessary to create all the state objects at once, new state objects may be created during runtime as needed.

1.10.6. Setting Input and Output

The process of setting input and output buffers for the features is the same as the non-Triton enabled SDKs. The only difference is that when the application is trying to send a batch of input (more than one video stream concurrently) to the server, it is necessary to provide batched buffers.

1.10.6.1 Batched NvCVImage Object Buffers

A batched NvCVImage points to a contiguous buffer that contains multiple images with the same structure. The Triton enabled SDK library will transfer NvCVImage to and back from the server using gRPC if the NvCVImage buffer is allocated on the CPU and using CUDA shared memory if the NvCVImage buffer is allocated on the GPU.

There are tighter restrictions on the type of images supplied to the Triton Client, relative to the native SDK:

- The pixel format is restricted to BGR, BGRA, and A; RGB and RGBA are not accommodated.
- The images must have a zero-gap pitch. This can be achieved by specifying 1 as the alignment, when calling the NvCVImage constructor or NvCVImage_Alloc, NvCVImage_Realloc.

1.10.6.2 Batch Utilities

The following utility functions in `BatchUtilities.cpp` help you to work with image batches:

- `AllocateBatchBuffer()` can be used to allocate a buffer for a batch of images.
- `NthImage()` can be used to set a view into the nth image in a batched buffer.
- `ComputeImageBytes()` can be used to determine the number of bytes for each image to advance the pixel pointer from one image to the next.
- `TransferToNthImage()` makes it easy to call `NvCVImage_Transfer()` to set one of the images in a batch.
- `TransferFromNthImage()` makes it easy to call `NvCVImage_Transfer()` to copy one of the images in a batch to a regular image.
- `TransferToBatchImage()` transfers multiple images from different locations to a batched image.
- `TransferFromBatchImage()` can be used to retrieve the images in a batch to different images in different locations.
- `TransferBatchImage()` transfers all images in a batch to another compatible batch of images.

The last three functions can also be accomplished by repeatedly calling the Nth image APIs, but the source code illustrates an alternative method of accessing images in a batch.

1.10.6.3 Allocation of Batched Buffers

To allocate batched buffers, call the `AllocateBatchBuffer()` function, which will allocate an image that is N times taller than the prototypical image.

Note

The allocation cannot always be interpreted this way, especially if the pixels are planar.

The purpose of this function is mainly to provide storage and then dispose of this storage when the NvCVImage goes out of scope or its destructor is called.

You can use your own method to allocate the storage for the batched images. The image that the `AllocateBatchBuffer()` yields is only used for bookkeeping and is never used in any of the SDK APIs.

Note

The SDK APIs require an NvCVImage descriptor for only the first image.

1.10.6.4 Setting the Batched Images

The API takes the image descriptors for only the first image in a batch. The following sample allocates the src and dst batch buffers and sets the input and outputs via the views of the first image in each batch buffer.

```
NvCVImage srcBatch, dstBatch, nthSrc, nthDst;
AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                    ...);
AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                    ...);
NthImage(0, srcHeight, &srcBatch, &nthSrc);
NthImage(0, dstHeight, &dstBatch, &nthDst);
NvVFX_SetImage(effect, NVVFX_INPUT_IMAGE, &nthSrc);
NvVFX_SetImage(effect, NVVFX_OUTPUT_IMAGE, &nthDst);
```

Because the image descriptors are copied into the SDK, this can be simplified to the following:

```
NvCVImage srcBatch, dstBatch, nth;
AllocateBatchBuffer(&srcBatch, batchSize, srcWidth, srcHeight,
                    ...);
AllocateBatchBuffer(&dstBatch, batchSize, dstWidth, dstHeight,
                    ...);
NvVFX_SetImage(effect, NVVFX_INPUT_IMAGE, NthImage(0, srcHeight, &srcBatch, &
↪nth));
NvVFX_SetImage(effect, NVVFX_OUTPUT_IMAGE, NthImage(0, dstHeight, &dstBatch, &
↪nth));
```

The other images in the batch are computed by advancing the pixel pointer by the size of each image.

The other aspect of setting the batched images is determining how to set the pixel values. Each image in the batch is accessible by calling the `NthImage()` function:

```
NthImage(n, imageHeight, &batchImage, &nthImage);
```

You can then use the same techniques that were used for other NvCVImages on the recently initialized nthImage view. As previously suggested, `NthImage()` is just a thin wrapper around `NvCVImage_InitView()` and can be used instead. The `NvCVImage_Transfer()` functions can be used to copy pixels to and from the batch.

1.10.7. Setting a Batch of State Objects

The SDK uses batch size parameter, NVVFX_BATCH_SIZE, to determine the number of items in the input/output buffers and the batch of states parameter, NVVFX_STATE, to determine which item in the batched input and output buffer belongs to which input video stream. It is necessary to set these parameters before the NvVFX_Run call.

A batch of states array is an array of NvVFX_StateObjectHandle pointers. In the following example, a batch size of 3 is used and a batch of states is created with pointers to the p-th, the q-th and the r-th items in the state_handles array. This signifies that all input/output buffers contain data in the same order. Since each state corresponds to an input video stream, it signifies that the 0-th, the 1-st, and the 2-nd items in the buffers belong to the p-th, the q-th and the r-th input video streams.

A batch can not contain more than one item from a single input video stream, so a batch of states should not have duplicate state objects.

Even when processing only one video, it is necessary to create a batch of states array with one item and set the batch size to 1.

```
unsigned batch_size = 3;

NvVFX_StateObjectHandle* batch_of_states[] = {&state_handles[p],
&state_handles[q], &state_handles[r]};

nvcv_err = NvVFX_SetU32(feature_handle, NVVFX_BATCH_SIZE, batch_size);

nvcv_err = NvVFX_SetStateObjectHandleArray(feature_handle, NVVFX_STATE,
batch_of_states);
```

The NVVFX_BATCH_SIZE and the NVVFX_STATE can be changed before every NvVFX_Run call.

Note

The size of the batch need not be equal to the total number of input video streams.

1.10.8. Running the Feature

NvVFX_Run runs the feature on the server. The call is asynchronous and non-blocking.

```
nvcv_err = NvVFX_Run(feature_handle);
```

1.10.9. Waiting for the Result

NvVFX_SynchronizeTriton will block the execution and wait until the results are available after the NvVFX_Run call. When this function exits, the results will be populated in the output buffers.

```
nvcv_err = NvVFX_SynchronizeTriton(feature_handle);
```

1.10.10. Destroying the State Objects

`NvVFX_DeallocateState` destroys the state objects.

```
for (int i=0; i<num_input_video_streams; i++) {

   nvcv_err = NvVFX_DeallocateState(feature_handle, state_handles[i]);

}
```

1.10.11. Destroying the Feature Instance

`NvVFX_Destroy` destroys the feature instance.

```
nvcv_err = NvVFX_Destroy(feature_handle);
```

1.10.12. Destroying the Triton Object

`NvVFX_DisconnectTritonServer` disconnects from the server and destroys the server object.

```
nvcv_err = NvVFX_DisconnectTritonServer(server_handle);
```

1.11. PCIe Multi-GPU systems

On multi-GPU systems, the Triton server uses peer-to-peer memory copy to transfer data between GPUs whenever this feature is available; that is, when `cudaDeviceCanAccessPeer()` returns true.

However, on bare-metal Linux systems with PCIe topology, IOMMU-enabled peer-to-peer memory copy is not supported. For more information, refer to Host IOMMU Hardware, PCI Access Control Services, and VMs. WHen IOMMU is enabled, we recommend setting it to passthrough (by setting the Linux kernel parameter `iommu=pt`) to ensure optimal performance.

The following are the steps to set IOMMU to passthrough in GRUB:

1. Determine whether IOMMU is enabled by running the following command:

   ```
   dmesg | grep -e DMAR -e IOMMU
   ```

   If the command produces no output, IOMMU is not enabled, and no further action is required.

   If the command produces output, IOMMU is enabled. Continue with the next step.

2. Determine whether IOMMU is set to passthrough by running the following command:

   ```
   dmesg | grep -i -e iommu=pt -e iommu.*passthrough
   ```

   If the command produces output, IOMMU is already set to passthrough, and no further action is required.

   If the command produces no output, continue with the following steps.

3. Open `/etc/default/grub` file for edit and add `iommu=pt` to `GRUB_CMDLINE_LINUX` option. For example:

```
.....

GRUB_CMDLINE_LINUX="crashkernel=auto quiet iommu=pt"

.....
```

4. Based on the system's OS, use `grub-mkconfig` or `grub2-mkconfig` to generate the configuration file:
   - On systems with BIOS:

     ```
     grub-mkconfig -o /boot/grub2/grub.cfg #on ubuntu, debian
     grub2-mkconfig -o /boot/grub2/grub.cfg #on centos, rockylinux
     ```

   - On systems with UEFI:

     ```
     grub-mkconfig -o /boot/efi/EFI/<os_name>/grub.cfg #on ubuntu, debian

     grub2-mkconfig -o /boot/efi/EFI/<os_name>/grub.cfg #on centos,
     ↪rockylinux

        #replace <os_name> with ubuntu, centos, debian, or rocky
     ```

5. On Ubuntu and Debian, you might need to install `grub-mkconfig`:

   ```
   apt install grub-common
   ```

6. Reboot the system:

   ```
   systemctl reboot
   ```

7. Verify that IOMMU is set to passthrough by repeating step 2.

Copyright